Automatic Roundtrip Engineering

Uwe Aßmann

Research Center for Integrational Software Engineering (RISE)
Programming Environments Lab (PELAB)
Linköpings Universitet, Sweden

Abstract

A systematic method for roundtrip engineering of systems, *automatic roundtrip engineering (ARE)*, is presented. It relies on the automatic derivation of inverses for domain transformations. While roundtrip engineering is a well known system engineering method, systematic conditions for its deployment have not yet been formalized, and this is done in the paper. Secondly, ARE is a generic architectural style for different architectural scenarios. To show this, the paper gives a first classification, defining several subclasses of ARE systems: sequenced ARE systems, automatic Model-View-Controller engineering (MVARE), and bidirectional aspect systems (*Beavers*). Sequenced ARE systems extend the ARE principle to chains of transformations. MVARE systems project a domain into a set of simpler ones, simplifying system understanding. Beaving systems generalize aspect-oriented programming to roundtrip engineering. All ARE classes describe different generic application architectures and have a great potential to simplify the construction of roundtrip engineering tools and applications.

1 Introduction

Usually, software systems are quite complex. If the developer has to deal with their full complexity, development is hard and costly. Hence, technologies to reduce software complexity are urgently desired. For this purpose, this paper proposes a new variant of roundtrip engineering, *automatic roundtrip engineering (ARE)*. ARE is a generic architectural style, defining conditions for the automatic derivation of composition and decomposition transformations in software development. While roundtrip engineering is widely used for software development [Int01], systematic conditions for an architectural style have never been defined. However, this is a prerequisite to support the technique with tools and automation.

Seen from a general perspective, roundtrip engineering is an instance of the method of *domain transformation* in mathematics and algorithmics. In a certain domain, if a problem is too hard to solve, a *domain transformation* is applied to

[1] Email: uweas@ida.liu.se

map the problem into a different domain. If in that domain a simpler algorithm exists, a solution is computed. By the *inverse domain transformation*, the solution in the original domain is found (Fig. 1). Since both problems and both solutions in the different domains are isomorphic to each other, users can choose in which domain they specify their problem, independently of in which domain the solution is computed.

Based on domain transformations, roundtrip engineering is a software development method, being supported by several integrated development environments (IDE) [Int01]. If an IDE supports roundtrip engineering, it maintains at least two representations of the software artefact, e.g., a textual and a graphical representation. Users can choose the representation in which they want to see and edit the artifact. By pressing a button, they can change the representation and continue to edit in the new representation. Since this works in roundtrip, the method is called *roundtrip engineering*. It has the decisive advantage that users can choose the appropriate form of editing for a certain development situation, e.g., they may use graphics when graphics explains the structure of the software better, or they may use text if text editing is faster. Of course, other representational domains can be supported, and roundtrip engineering works with several domains as well. However, so far roundtrip engineering systems are being constructed in an ad-hoc manner and on an individual basis; no systematic method is known to derive the implementations.

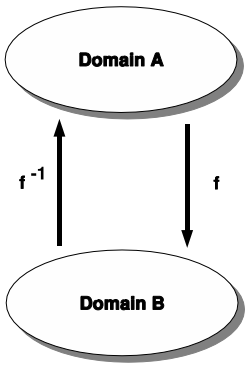

Figure 1. Two domains with their domain transformations.

This paper proposes a new, systematic method to construct roundtrip engineering systems, *Automatic Roundtrip Engineering (ARE)*. ARE can be regarded as an architectural style, as a paradigm to construct software, as a design pattern for software systems, as a reference system architecture for software development environments, or as a system development method. Its central idea is very simple (Fig. 2). Given a specification of a forward domain transformation or a projection, how can we derive automatically an inverse, such that roundtrip between different representations is possible automatically?

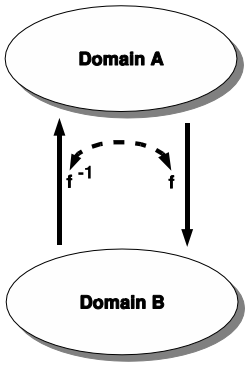

Figure 2. A simple instance of an ARE.

If more than 2 domains are involved, roundtrip engineering comes in another flavor. A domain transformation may be a domain decomposition, i.e., it decomposes the specification into two or more simpler domains (*views, dimensions, concerns,* or *aspects*), which contain partial knowledge about the full specification (Figure 3). In mathematics, such decomposition functions are called *projections*. As soon as projections are available, a software systems can be decomposed into simpler *parts*. If there is also a simple integration method, reconstructing the whole becomes easy.

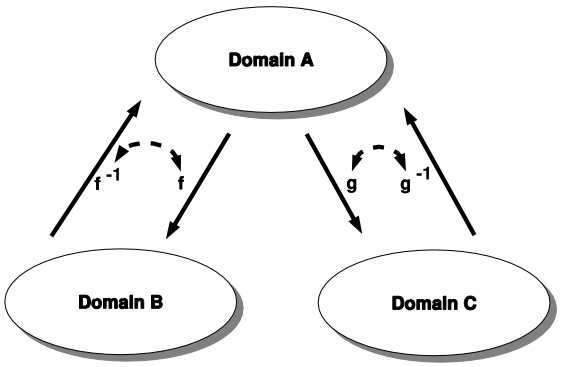

Figure 3. A simple instance of a model-view-controller based ARE. Projections transfer a complex domain to more simpler domains.

For projections and decompositions, a special variant of ARE results, *Model-View-Controller based engineering (MVARE)*. Its basic idea is to *decompose* software into simpler forms, and to compute a reintegration function automatically. Applying decomposition and reintegration, an automatic roundtrip becomes possible. This process allows for changing the representation of the system into simpler projections easily. In this architectural substyle of ARE, decomposition and compo-

sition follows the Model-View-Controller design pattern [GHJV94]: the software system is regarded as the *model* (i.e., as the *data* which is going to be constructed), there are one or many *views* in which the model is presented and edited, and there is a controller that maintains the relations between views and model, ensuring that every change in every view is propagated into the model and into the other views. In essence, MVARE applies the MVC pattern to software construction, regards the code as the data that is being evolved in an interactive process. Therefore, MVARE is also an entirely incremental process.

2 Automatic Roundtrips

It is the purpose of this paper to give systematic definitions for a roundtrip-based system, and this section defines two simple forms of automatic roundtrip engineering (ARE) systems.

Definition 2.1 Let A, B be two domains, and $f : A \to B$ a transformation function from function space F. If there is a functional $i : F \to F$ which calculates for f its inverse $f^{-1} \in F$ then $R = (A, B, f, i)$ is an *automatic roundtrip system (ARE)*.

Usually, i is not unique. There may be many algorithms that invert an f; however, it must hold $i \cdot f = id$.

Example 2.2 *XSLT.* As a example of an ARE system, consider an XSLT transformation f from xml to html, which is invertible under the XSLT script inversion functional i. Then, $(xml, html, f, i)$ is a ARE system.

Decompilers. A decompiler that reconstructs the source of a program forms together with its compiler an ARE system, if the decompiler is automatically constructed from the compiler.

Many tools are roundtrip engineering tools, but the inverse transformation is hand-coded, and not automatically derived. For instance, *refactoring tools* that read a software package and write it in the same form to file again, are roundtrip-based tools. Refactoring retains syntax and semantics for all parts of the package which are not modified (*source code hygiene* [LH01]). This behavior is required since the result of the refactoring should be evolved further, and this is easiest if nothing changed of the package except the refactored parts. Because refactoring tools read and write a software package in the same form, they form instances of roundtrip engineering. Examples are tools such as Recoder [LH01], CodeMorpher [XPT01] or Together [Int01].

Also, ARE systems can be stacked onto each other (Fig. 4):

Definition 2.3 A *sequenced ARE system* is a sequence of ARE systems $R_1, \ldots, R_n$, in which source and target domains are pairwise compatible, i.e., it holds $B(R_j) = A(R_{j+i})$, for $j = 1, \ldots, n-1$.

Example 2.4 *XSLT chains.* As an example, consider two XSLT transformations f_1 and f_2, which are invertible under the XSLT script inversion functionals i_1, i_2.

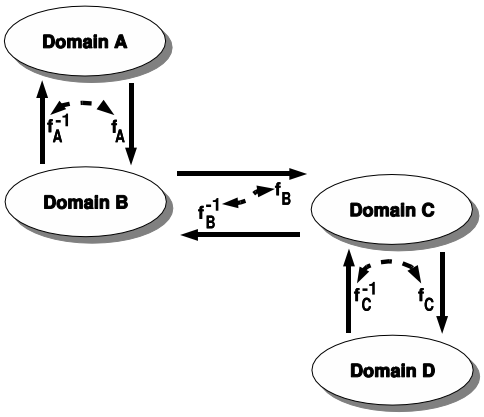

Figure 4. A sequenced ARE.

Then, $((A, B, f_1, i_1), (B, C, f_2, i_2))$ is a sequenced ARE system. Such systems are very useful in Web Engineering. Usually, in a content management system, chains of XSLT transformations are employed to calculate the final view on a data base or web site. If the sequence of XSLT transformations is an ARE, users may edit the final view, and the ARE can update the database or web site automatically.

Higherings and Lowerings of Intermediate Representations. A compiler or a transformer maintains several intermediate representations, usually, a high, a medium, and a low one (*HIR*, *MIR*, *LIR*). Between those, the compiler performs *highering and lowering operations*; it *lowers* from the HIR to the MIR to the LIR, and *highers* the other way. On the first sight, one might wonder why a compiler should higher, but this is necessary for good optimization. On each level of abstraction, different optimizations are possible, and optimizations on lower levels enable more optimizations on higher levels. When an optimization on the MIR has been performed, the compiler highers the information to enable more optimizations on the HIR. For instance, after dead code elimination on the MIR, more array loop optimizations on the HIR are possible. Hence, highly optimizing compilers need to lower and to higher. Highering and lowering, however, are inverse transformations. If they can be derived automatically from each other, a compiler is an ARE system.

Sequenced ARE systems bridge gaps between domains that are semantically very much distinct. Often, direct inverses to functions cannot be given if the transformational gap between the domains is too large. Then, as a general best practice, the tool builder introduces more intermediate representations, and splits the function into a sequence of simpler ones, which may be invertible. Hence, sequenced ARE systems can be employed in situations where the domains that should be mapped to each other are pretty different.

These examples illustrate that ARE-based systems have the advantage that users can choose in which domain they want to look at the software. In every circumstance, the ARE system can map the actions of the user into the other domain. ARE systems are perfectly apt for interactive applications in which several views of the

same software artefact exist. Users can choose the view on the software and switch it on demand.

3 MVARE - Automatic Model View Controller Engineering

Automatic Model View Controller Based Engineering (MVARE) is a variant of ARE in which several invertible transformations to different target domains exist (Fig. 3). These target domains represent the source domain partially such that a *model* and several *views* can be distinguished.

Definition 3.1 Let $A, B_1, \ldots, B_n$ be $n+1$ domains, and $f_j : A \to B_j$ a transformation function from function space $F, j = 1, \ldots, n+1$. If there is a functional $i : F \to F$ which calculates for every f_j its inverse $f_j^{-1} \in F$ then $R = (A, B, f_1 \ldots f_n, i)$ is an *automatic MVC roundtrip* system *(MVARE)*. A is called the *model*, while $B_1, \ldots B_n$ are called the *views*. The f_j are called the *projections*, the f_j^{-1} are called the *integrations*.

As above, i need not unique. In the MVARE architectural style, the model can be recomputed if a view changes, and vice versa. Unfortunately, for MVARE, we cannot give too many complete, but only semi-complete examples.

Example 3.2 *Projections/Integrations.* In view-based programming, software is specified with *views*. In a view, certain parts of the model are hidden, while others are visible. The system results by integrating all views. To this end, several programming languages have been developed (Meld [KG87], CoSy-fSDL [WKD94], hyperslices [TOHS99]). However, usually they do not allow for decomposition of the system to the views.

UML Tools. Although MVC-based roundtrip systems exist, the roundtrip is not automatic. Consider, for instance, Together, which can read legacy sources and display them in UML structure diagrams [Int01]. Together permits edit either the diagrams or the source, and keeps both forms synchronized. After a change in a diagram the source is regenerated, while after a change in the source, the diagrams are recomputed. Since several forms of diagrams are possible, the system is projected into several UML views. The code generation reintegrates them again. Together's roundtrip engineering is one of its main advantages, however, it is not achieved as a result of a systematic specification method or architectural style.

MVARE With DPO Graph Rewriting. [RBL02] defines the above criteria if the transformations are specified with double-pushout graph rewriting (DPO), resulting in a special variant of MVARE, called CODEX. A DPO-based graph rewrite system is invertible. If every projection f_j is described as a such a system, the integrations f_j^{-1} result automatically. However, during integration, special care has to be taken, in order to deal with the dependencies between all transformations. And this dependency theory is the heart of CODEX.

MVARE systems support divide-and-conquer based system development. With hand-coding, view-based roundtrip systems are hard to construct. If the projecting

transformations are invertible, however, integrations can be automatically derived and the construction of view-based roundtrip systems becomes simple.

4 Bidirectional Weavers (Beavers)

The principle of automatic roundtrip engineering can also be applied to aspect-oriented programming (AOP). AOP is somewhat similar to view-based programming, however, distinguishes a *core part* of the system that carries the major functionality, from *aspects* that provide additional functionality or non-functional qualities. Also in AOP, integrations exist: *weavers* weave aspect specifications into the core and emit the system. In the following, we focus on *static weaving*, i.e., on transformational weaving at compile-time. Aspects can also be woven dynamically [KLa01].

In AOP, static weavers can be regarded distribution operators that distribute aspect fragments over core code [AL99]. Usually, AOP weavers only integrate aspects into the core, but do not deweave aspects from an integrated system. This would be the task of a *deweaver:* deweavers extract code from code, projecting complex integrated systems into aspects.

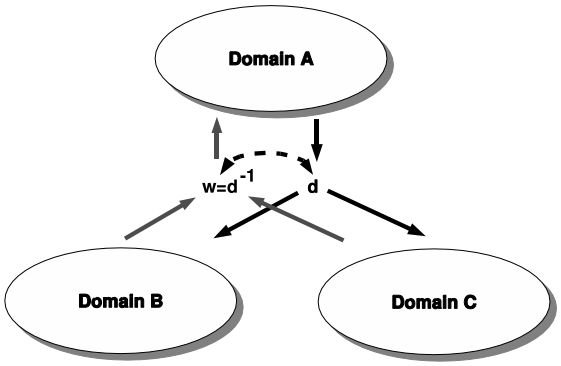

Figure 5. A weaver and its inverse, the deweaver, seen as inverse functions.

If, from a weaver, a deweaver can be derived automatically, AOP becomes an instance of ARE: The weaver implements the integration function, while the deweaver implements the projection function. Such an AOP system is called a *bidirectional AOP* system, and the weaver a *bidirectional weaver:*

Definition 4.1 Let $A, B_1, \ldots, B_n$ be $n+1$ domains, and $d : A \to B_1 \times \ldots \times B_n$ a transformation function from function space F (a *deweaver*). If there is a functional $i : F \to F$ which calculates for every d its inverse $w = d^{-1} : B_1 \times \ldots \times B_n \to A$ (a *weaver) then* $R = (A, B, d, i)$ is an *bidirectional AOP* system *(Bi-AOP).* The function pair (d, d^{-1}) is called a *bidirectional weaver (beaver).*

Example 4.2 *Debugging of AOP systems.* Debugging of a weaved system is usu-

ally a bit problematic, since the weaver does not generate *debug information* for aspects. With such information, users could trace back program fragments in the system to aspects. In particular, they would be enabled during debugging to identify to which aspect a variable or a program fragment belongs. However, this trace-back is the same process as inverting the weaving, and if the debug information is rich enough, a deweaver could use the information to recompute the aspects automatically. Usually, AOP systems do not support this.

If d can be split into $f_1, \ldots, f_n$, and w can be split into $f_1^{-1}, \ldots, f_n^{-1}$, and a bidirectional AOP system is an MVARE. Then, the weaver implements the integrations f_j^{-1}, while the deweaver implements the projections f_j. However, usually, however, both weaver and deweaver do not work pointwise because they treat tangling and cross-cutting of aspects over cores.

5 Conclusion

This paper is dedicated to Gerhard Goos. On many occasions, we have discussed how to unify TEX and emacs. Clearly, this requires a full roundtrip system for TEX, i.e., a system in which TEX text can be transformed to bitmaps and vice-versa, without loosing information. Then, in every representation, edits are possible and users can switch the view by pressing a button.

This paper contains a solution to this problem in terms of a formally defined architectural style: if TEX was a (sequenced) ARE system, the user could edit either representation and switch to the other representation losslessly. Suppose, the TEX compiler consists of several transformations, for each of them an inverse should exist. Then, the decompiler that maps bitmaps back to the source, would result as a concatenation of all inverses in the sequenced ARE architecture. Even more, every tool that attempts such an endeavour, must, in the end, be a roundtrip system. And it is not too bold to say that every tool that should be developed as simply as possible, should be an ARE-based system. It is hoped that future versions of TEX can be constructed in this way.

References

[AL99] Uwe Aßmann and Andreas Ludwig. Aspect weaving by graph rewriting. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, volume 1799 of *Lecture Notes in Computer Science*, pages 24–36. Springer, Heidelberg, October 1999.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[Int01] Object International. Together case tool, 2001. Version 5.0.

[KG87] Gail E. Kaiser and David Garlan. MELDing data flow and object-oriented programming. In *Proceedings of OOPSLA'87*, number 22(12) in ACM SIGPLAN Notices, pages 254–267. ACM Press, New York, 1987.

[KLa01] Gregor Kiczales, Christina Videira Lopes, and al. Getting started with Aspect-J. *Communications of the ACM*, 44(10):59–65, October 2001. http://www.aspectj.org.

[LH01] Andreas Ludwig and Dirk Heuzeroth. Meta-programming in the large. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, volume 2177 of *Lecture Notes in Computer Science*, pages 443–452. Springer, Heidelberg, October 2001.

[RBL02] Kalle Ring-Burbeck and Henrik Larsson. Automatic Model View Controller Engineering. Master's thesis, Linköpings Universitet, June 2002.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M., Jr. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of ICSE'99*, pages 107–119, Los Angeles CA, USA, 1999.

[WKD94] H.R. Walters, J.F.Th. Kamperman, and T.B. Dinesh. An extensible language for the generation of parallel data manipulation and control packages. In P. A. Fritzson, editor, *Proceedings of the Poster Session of Compiler Construction*, number LiTH-IDA-R-94-11 in Research Reports. Linköping University, Sweden, 1994.

[XPT01] XPTools. CodeMorpher refactoring tool, 2001. Version 1.0, http://www.xptools.com.